Sistemas Informáticos

Curso 2005-2006

PUMUKI
The Pure Music Kit

Pedro Javier Calle Cantalapiedra

Javier Chávarri Álvarez

Fernando Iglesias Pulido

Dirigido por:
Prof. D. Jaime Sánchez Hernández
Dpto. Sistemas Informáticos y Programación

Facultad de Informática

Universidad Complutense de Madrid

Resumen:
PUre MUsic KIt es una herramienta software de composición musical multicanal, que permite al usuario crear sus propias melodías usando sonidos de instrumentos generados mediante diversas técnicas (modelado físico, síntesis FM, ...). Utiliza el ratón y el teclado del PC como interfaz para esta generación de sonido, recogiendo los eventos de estos dos dispositivos para transformarlos en modificaciones de los parámetros del instrumento que esté sonando: volumen, nota actual, intensidad, *vibrato*, ... El uso del ratón y el teclado como interfaz de entrada al sistema aumenta enormemente la versatilidad de este software y el número de usuarios potenciales.

Abstract:
PUre MUsic KIt is a multichannel music composition software tool, that provides user a way of creating his or her own melodies using instruments sounds that are generated through many techniques (physical modeling, FM synthesis, ...). The application uses mouse and keyboard as an input interface to this sound generation, translating events read on these devices into parameters of the instrument that is sounding: volume, actual note, *vibrato*, ... This input interface increase hugely the versatility of this software and the potential audience of it.

Listado de palabras clave para búsqueda bibliográfica
música, sonido, interpretación, instrumento, síntesis, polifonía, modelado

Índice general

Agradecimientos

Me gustaría agradecer a mis amigos toda la paciencia que tienen cuando me pongo a hablar de mis libros, rol, programación, PUMUKI,... incluso que se muestren interesados. A mi abuelo por aguantar mis venturas y desventuras, a mi abuela y a Paloma por enseñarme a reaccionar y a ser como soy, y a Jose Manuel por enseñarme que nuestro tiempo nunca pasará.

Fernando Iglesias

A Ramón, Arbi y Javilón, por tener que aguantar en demasiadas ocasiones a tres informáticos desarrollando con todo lo que ello acarrea (que no es poco), no solo sin quejarse sino encima de buen humor. A todos los compañeros que han estado ahí para cualquier duda, problema,... A Fer y Pedro (¡por muchas duomos más!). A mi familia, por no haberles podido ir a ver este año todo lo que me habría gustado, y por su apoyo constante e indispensable.

Javier Chávarri

A mi padres, que este año me han visto parar muy poco por casa, y les hubiera gustado haberme visto mucho más. A mi abuelo, que le hubiera gustado verme entregar el presente, y verme acabar algo que llevo tanto tiempo persiguiendo. A Merce, por las innumerables tardes viéndome trabajar con el portátil, y a la que me gustaría haber dedicado mucho más tiempo. Y a mis dos compañeros de proyecto: Javi y Fer, que me han hecho disfrutar del compañerismo, y que me han hecho ver que con un buen ambiente, se es capaz de estar trabajando mucho y bien.

Pedro Javier Calle

A Jaime por habernos aguantado en las reuniones de este proyecto y nuestras discusiones sobre vi y emacs.

Pumuki Developers Team

Capítulo 1

Prefacio

La síntesis de sonido trata de generar sonidos que sean:

- Musicalmente interesantes
- Preferiblemente realistas (que suenen como algún instrumento real)
- Producidos en tiempo real

Una posible taxonomía de la síntesis de sonido digital podría ser la siguiente [7]:

1. Algoritmos abstractos
2. Grabaciones preprocesadas (sampling)
3. Síntesis espectral
4. Modelado físico

Comenzaremos hablando sobre los tres primeros para, finalmente, centrarnos en el cuarto punto.

1.1. Algoritmos abstractos

Las características de este tipo de algoritmos son la simplicidad y facilidad de implementación. El sonido producido es más o menos artificial comparado con otros métodos más complejos y sofisticados.

1.1.1. Frecuencia Modulada (FM) ('70)

Los primeros sintetizadores digitales y chips de tarjetas de sonido sintetizadoras están basadas en FM.

- Muy simple, fácil de implementar, y sólo con un par de osciladores de control de voltaje (VCOs).

- Estructura variable en el tiempo, como en los sonidos naturales.
- Timbres metálicos.
- Se consigue sonido armónico cuando el cociente entre la portadora y el modulador es un valor entero.
- Los sistemas realimentados añaden estabilidad al comportamiento en frecuencia.

1.1.2. Síntesis de Forma de Onda (Distorsión no-lineal) (Finales de los 60)

- Se le aplican funciones de forma no lineal a la señal de entrada.
- En la mayoría de los casos esta función de excitación es una señal sinusoidal.
- Usando una combinación de los polinomios de Chebyshev [9], los cocientes de los armónicos se pueden controlar. Por tanto, se puede hacer corresponder con la estructura armónica de un instrumento real.
- Se puede aplicar post-procesado, como puede ser el caso de la modulación en amplitud (AM).
- La señal de excitación puede ser otra además de la sinusoidal.

1.1.3. Algoritmo de Karplus-Strong (1983)

- Muy simple y, computacionalmente, un algoritmo útil.
- Usa un pequeño buffer de sonido, inicializado con datos aleatorios.
- Es un buffer circular, filtrado por un filtro paso-baja [10] después de cada lectura.
- Se consiguen tonos de cuerda punteada o de percusión.
- Se puede implementar sólo con operaciones de suma y desplazamiento.

1.2. Grabaciones preprocesadas (sampling)

La manipulación de sonidos grabados data de antes de 1920. Los requerimientos de memoria han supuesto siempre un problema para la síntesis de sampleo digital. La idea es grabar sonidos en muestras relativamente pequeñas, que serán los posteriormente reproducidos.

La primera de las dos figuras es una operación de filtrado simple, resultando un tono de punteado de cuerda. Por otro lado, la segunda figura

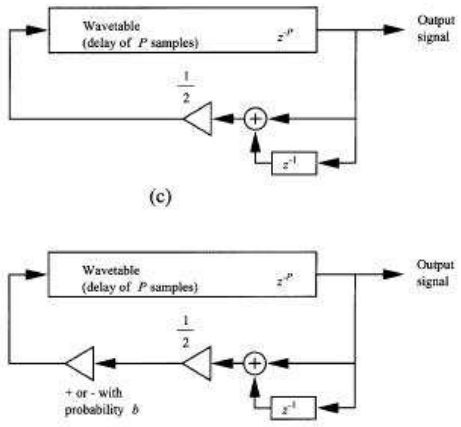

Figura 1.1: Algoritmo de Karplus-Strong.

cambia el signo de cada uno de los bits filtrados, con una cierta probabilidad. Ésto produce un tono de percusión.

1.2.1. Síntesis digital con tabla de ondas

Se usa una tabla donde se almacenan diversos muestreos de los instrumentos sintetizados. Almacenada en chips ROM o en disco, la tabla de ondas proporciona un sonido digitalizado real de un instrumento, que la adaptadora de audio puede manipular según se requiera. Un sintetizador de tabla de ondas puede tomar una muestra de un instrumento tocando una sola nota y modificar la frecuencia para tocar cualquier nota de la escala.

Algunas variedades de síntesis digital usando tabla de ondas:

1. Multiple Wavetable synthesis
2. Síntesis Granular (1940s)
   - *a*) Asíncrono (AGS) (1990s)
   - *b*) Pitch Síncrono (PSGS) (1990s)

1.3. Síntesis espectral

La idea es que almacenamos las propiedades de sonido percibidas.

1.3.1. Síntesis Aditiva

- Sumando componentes sinusoidales de diferentes fases, amplitudes y frecuencias.

- Las funciones de control de una sinusoide simple tienen una variación lenta.
- Desventaja: gran cantidad de datos (parámetros de control) y gran número de osciladores.

1.3.2. Fase codificadora de voz

- Se puede ver como un banco de filtros o un analizador STFT[1].
- Escala de tiempo y transposición sencilla del tono:
  - Escala de tiempo acoplada por modificación de la diferencia de tiempo de *frames* sintetizados consecutivos.
  - Modificación del tono modificando la escala de tiempos, y entonces cambiar la frecuencia de muestreo de la señal.
- Funciona para armónicos, los tonos varían lentamente.

1.3.3. Síntesis de Filtrado-fuente

- La señal de excitación se filtra con un filtro de variación de tiempo.
- También es llamado *Síntesis Substractiva*: la señal de entrada filtrada con un espectro de ganancia armónica.
- No representa un sonido muy realista.
- Usada en sintetizadores analógicos.

1.3.4. McAulay-Quatieri (MQ) (1986)

- La señal original se descompone en un conjunto de sinusoides.
- Análisis: STFT, detección de pico.
- Síntesis: interpolación de trayectoria, síntesis aditiva.

1.3.5. Síntesis de Modelo Espectral (SMS) (finales de los 80)

- Análisis sinusoidal con MQ $\Rightarrow$ parte de la señal determinista.
- Modificación sencilla de la duración (*tempo*) y frecuencia (*clave*).

[1] Short Time Fourier Transform

1.3.6. Síntesis de Modelo Transitorio (TMS) (1997)

- Es una extensión de SMS.
- La parte de *ruido* es sólo de los componentes de asentamiento ruidosos.
- La dualidad tiempo $\longleftrightarrow$ dominio de frecuencia: los impulsos de señales en dominio del tiempo $\longleftrightarrow$ sinusoide en dominio de frecuencia.
- DCT produce una sinusoide real-evaluada cuando la señal en dominio del tiempo es un pulso.
- Los transitorios detectados y parametrizados, son resintetizados $\Rightarrow$ faltan los componentes de ruido de asentamiento.

1.3.7. Síntesis FFT^{-1} (1992)

- Puede ser visto como síntesis aditiva en dominio de frecuencia.
- Todos los componentes de la señal se añaden juntos envueltos en un espectro.
- Posibilita la síntesis de sonidos complejos.

1.3.8. Síntesis Formante

- Formante es una concentración de energía en un espectro *energía/fuerza*.
- Originalmente para procesado de conversaciones.
- Se usa en la síntesis de conversaciones y en canciones.

1.4. Modelado físico

Nuestra aplicación hace uso de modelado físico para generar el sonido, pero ¿qué es el modelado físico?, intentaremos dar una idea intuitiva. Es una aproximación a la síntesis de sonido usando computadoras, simplemente consiste en una simulación de la estructura física de un instrumento por medio de fórmulas. En nuestro caso, nos fijaremos en la modelización de la vibración de una cuerda, por medio de la ecuación de onda.

1.4.1. Ecuación de onda de una cuerda

Para comenzar, vamos a suponer una cuerda sujeta por ambos extremos. Para hacer un análisis más exacto, necesitamos tener en cuenta que la variable y está asociada a un tiempo t y una posición x a lo largo de la cuerda. Como y estará asociada a dos variables, lo apropiado sería que las ecuaciones que la modelizan se realizaran en *derivadas parciales*.

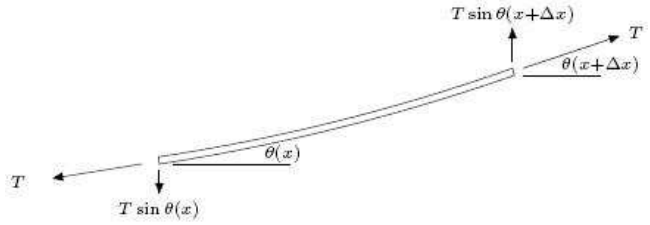

Figura 1.2: Esquema de una cuerda

Así, la ecuación que describe la vibración de una cuerda se llama *ecuación de onda* en una dimensión, y que es lo que ahora vamos a tratar.

Hay que tener en cuenta que, por simplicidad, sólo vamos a tener en cuenta las *ondas transversales*, que son las perpendiculares al movimiento de la cuerda; y *ondas longitudinales*, que irán en paralelo a la cuerda, y que no las tendremos en cuenta para el análisis.

Siendo T la tensión de la cuerda(en $newtons = \frac{kgm}{s^2}$), y ρ para la densidad lineal de la cuerda (en $\frac{kg}{m}$). Por lo que la posicion x a lo largo de la cuerda, el ángulo $\theta(x)$ entre la cuerda y la horizontal satisfará $tan\theta(x)=\frac{\partial y}{\partial x}$.

Por lo que, en otras palablas, con tal que $\theta(x)$ no se alargue mucho, el movimiento de la cuerda, esencialmente, se modela por esta ecuación de onda:

$$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2} \tag{1.1}$$

donde $c = \sqrt{T/\rho}$.

D'Alembert descubrió un método muy simple para encontrar una solución general a esta ecuación. A grandes rasgos, esta idea era factorizar el operador diferencial mediante un doble cambio de variables; por lo que, finalmente, nos queda una expresión así:

$$\frac{\partial^2 y}{\partial u \partial v} = 0 \tag{1.2}$$

Esta ecuación se puede ver directamente que tiene como solución general la dada por $y = f(u) + g(v)$ para f y g dos funciones que encajen correctamente. Sustituyendo en la ecuación, obtenemos:

$$y = f(x + ct) + g(x - ct) \tag{1.3}$$

Esto representa la superposición de dos ondas, una viajando hacia la izquierda y otra hacia la derecha, ambas con una velocidad c.

Físicamente, esto significa que la onda que viaja a la izquierda choca con el final de la cuerda y vuelve invertida, como una onda que va hacia la derecha. Éste es el *Principio de reflexión* [8].

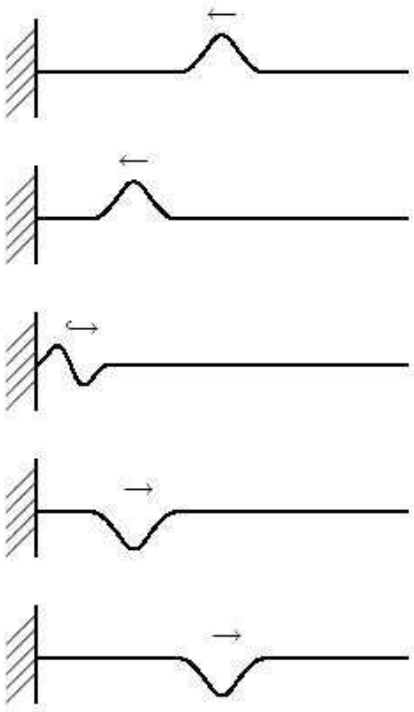

Figura 1.3: Evolución de una onda en una cuerda

1.4.2. Tipos de modelos físicos

Podemos encontrar dos grandes aproximaciones con algunos programas que las usan.

Modelo de masas

Parte de simular el material por donde se va a desplazar la onda como una sucesión de masas idénticas y equidistantes unidas por muelles de masa despreciable. Si provocamos que la primera masa se desplace hacia la segunda, el muelle entre ambas se comprime e intenta volver a su posición de reposo, ejerciendo una fuerza hacia sus dos extremos que es proporcional a cuanto está comprimido el muelle. Debido a esta fuerza la siguiente masa se desplaza a su derecha provocando el mismo efecto sobre el muelle, se repite este proceso hasta que alcanzamos la última masa. La velocidad de propagación de la onda viene determinada por las propiedades del material. Si hacemos que se desplace una masa que no sea de uno de los extremos se generan dos ondas en direcciones opuestas que rebotan en los extremos y regresan a la masa que las generó. Algunos programas que usan esta aproximación son TAO [4] y PureData [5].

Modelo de Karplus-Strong

El sistema que se considera es el siguiente, tenemos un retardo y un filtro. La salida del filtro es la salida del sistema y vale para realimentar la entrada del retardo. El retardo se inicializa con *ruido blanco* y la salida de éste es la entrada del filtro. Hay componentes en frecuencia que se atenuarán más rápidamente que otras, por lo que la entrada del retardo se hará progresivamente una señal pseudo-periódica en la que se puede apreciar claramente su frecuencia fundamental. La magnitud de respuesta del filtro no debe exceder una unidad para cualquier frecuencia, caso de hacerlo el sistema se volverá inestable. Uno de los problemas de este modelo es el siguiente, la frecuencia fundamental del tono no puede ser controlada con precisión, esto es, la frecuencia fundamental f_0 de la señal generada está determinada por la longitud del retardo, a saber

$$f_0 = \frac{f_s}{M + 1/2}$$

donde f_s es la frecuencia de muestreo (en Hz), M es la longitud del retardo y 1/2 es un factor introducido por el filtro. Una librería que usa esta aproximación es STK [3].

Capítulo 2

Manual de PUMUKI

2.1. Introducción

2.1.1. ¿Qué es PUMUKI?

PUMUKI pretende acercar la generación de música usando síntesis de sonidos avanzadas a cualquier usuario de PC. Para ello utiliza una interfaz tan genérica y común como son el teclado y el ratón de cualquier ordenador como medio para generar composiciones por el usuario con una calidad de sonido muy similar a los instrumentos reales.

2.1.2. Motivación

La idea de PUMUKI nace por la dificultad para cualquier usuario no avanzado en el uso de programas secuenciadores de comenzar a crear con unos medios mínimos. Cualquier software musical de hoy en día que proporcione unas mínimas funcionalidades tiene unos requisitos que muchas veces para el usuario *no iniciado* se convierten en insalvables cuando se quieren obtener una calidad de sonido medianamente aceptable. Secuenciadores como Cubase, Reason, ... necesitan algún tipo de interfaz MIDI como teclados, además de requerir algún tipo de tarjeta de sonido semiprofesional para poder utilizar sonidos que se basen en modelado físico o síntesis de sonido de *alto nivel*.

Nuestra idea inicial fue intentar conseguir una herramienta que permitiese generar sonido de alta calidad sin el *handicap* que suponen los requisitos anteriormente mencionados. Para ello usamos la interfaz más común y genérica que se conoce (el teclado y el ratón del PC) para producir la música que el usuario desea componer.

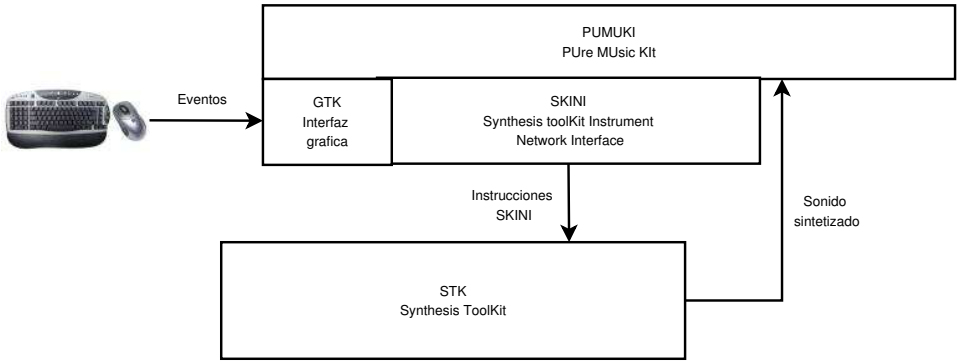

Figura 2.1: Diagrama de componentes de PUMUKI

2.1.3. ¿Qué hace PUMUKI?

PUMUKI recoge eventos externos del teclado y el ratón del PC para convertirlos en sonido de instrumentos. Este proceso de conversión se apoya en varios componentes que definimos a continuación:

- *Interfaz gráfica (GTK):* Recoge los eventos tanto de ratón como de teclado utilizando las funciones que incorporan las librerías GTK, tomando los datos necesarios para generar las instrucciones SKINI: tecla pulsada, posición del ratón, ... El uso que hacemos de estas librerías gráficas en este proyecto se detalla en el capítulo 3, página 21.

- *Instrucciones para las librerías de sonido (SKINI):* PUMUKI transforma los eventos recogidos a través del teclado y el ratón en instrucciones con formato SKINI, modificando parámetros del sonido que actualmente esté en uso. Si se trata de un violín, por ejemplo, podríamos cambiar la nota, el volumen o la inclinación del arco. El conjunto de instrucciones de control SKINI es analizado con más detenimiento en el capítulo 4, página 24.

- *Librerías de síntesis (STK):* Las librerías STK reciben instrucciones de control SKINI en tiempo real, y producen el cambio indicado por éstas en el sonido generado. Se expone las funcionalidades y el uso que hace PUMUKI de estas librerías en el capítulo 5, página 26.

La interfaz gráfica de PUMUKI permite grabar, reproducir y modificar proyectos con hasta 4 canales. A cada canal se le puede asociar un valor de reverberación distinto, tanto en grabación como en reproducción. Además, cada canal puede tener un instrumento distinto asociado.

2.2. Instalación

PUMUKI por ahora solo está disponible en plataformas Linux. Por tanto, primeramente necesitaremos tener un PC con alguna distribución de Linux instalada.

2.2.1. Requisitos de PUMUKI

Para la instalación de PUMUKI además es necesario tener instaladas las librerías STK antes. Estas librerías se pueden obtener de la siguiente url:

`http://ccrma.stanford.edu/software/stk/download.html`

Una vez bajados los fuentes de STK, debemos compilarlos tal y como se indica en el fichero README dentro del archivo tar.gz.

NOTA: Estas librerías a su vez tienen otras dependencias. Para resolverlas, se aconseja leer el fichero README y la documentación que se incluye en directorio `doc` dentro del directorio de las librerías STK.

Además, necesitaremos las librerías gráficas GTK. Podemos obtenerlas bajando los paquetes `libgtk2.0-0` y `libgtk2.0-dev`.

2.2.2. Instalación de PUMUKI

Primeramente descomprimimos el archivo PUMUKI.tar.gz. Antes de ejecutar la aplicación habrá que copiar el archivo `demo.cpp` que se incluye en la carpeta `pumuki` dentro del directorio de las librerías STK y recompilarlas de nuevo.

Además, para poder compilar el proyecto, se deben incluir en el directorio `/usr/include/stk` todos los archivos que estén en el directorio `include` que está en el directorio donde estén instaladas las STK.

Debemos configurar el archivo `pumukiExec` para que apunte al directorio donde tengamos instaladas las STK. Por ejemplo, si están instaladas en /home/sik/stk-4.2.1 el archivo pumukiExec debe contener la siguiente información:

```
./bin/pumuki | /home/sik/stk-4.2.1/projects/demo/demo Bowed -n
                              4 -or -ip
```

Compilamos tecleando `make`.

Ahora podemos ejecutar PUMUKI tecleando `./pumukiExec` dentro de la carpeta `pumuki`.

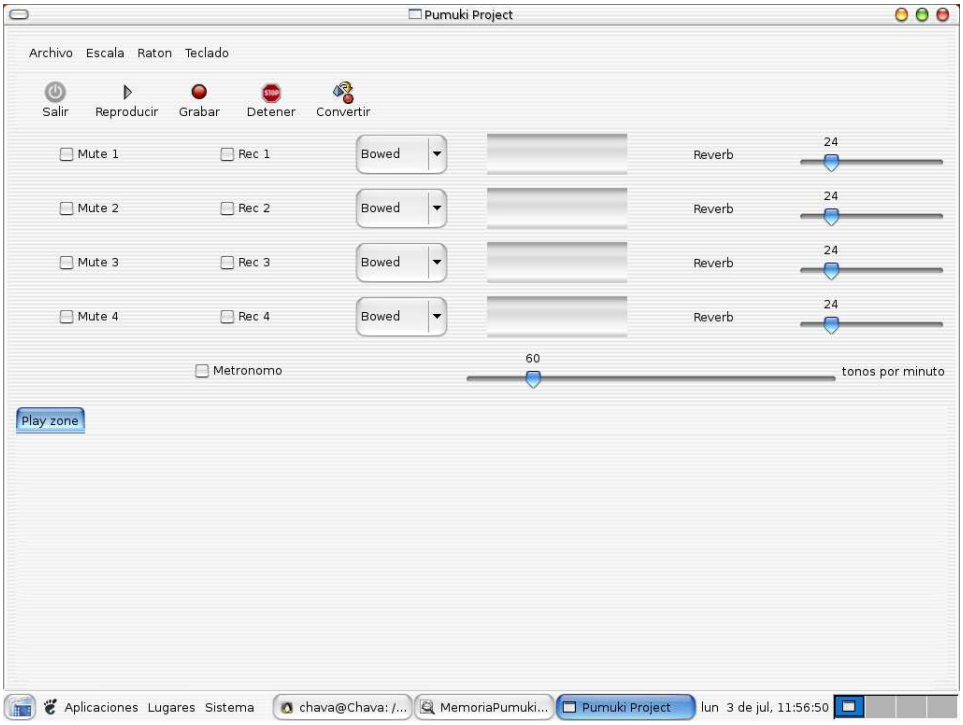

Figura 2.2: Pantalla inicial de PUMUKI

2.3. Empezando...

Al ejecutar PUMUKI, podemos ver la pantalla que se muestra en la figura 2.2.

PUMUKI arranca siempre con un proyecto vacío por defecto. Esto quiere decir que en principio ninguna pista contendrá sonido alguno, y que si queremos arrancar de algún proyecto ya hecho, deberemos cargarlo antes de empezar a trabajar con él. El procedimiento para cargar un proyecto ya creado se detalla en la sección 2.4, página 19.

Veamos por tanto cómo empezar a grabar nuestra primera pista. El teclado de PUMUKI está mapeado a notas musicales tal y como se indica en la figura 2.3. El movimiento de ratón nos dará el volumen con el que sonará nuestro instrumento. Si los movimientos son más rápidos aumentará el volumen del sonido del instrumento, y a la inversa si disminuye.

La zona etiquetada como *Play zone* es la parte de la pantalla donde se recogerán los movimientos del ratón. Fuera de esta zona el volumen no cambiará a pesar de que el ratón esté en movimiento.

Para grabar en una pista debemos seleccionar la casilla *Rec* de la pista en que queramos grabar, y después pulsar el botón *Grabar* que se muestra en el panel de botones de la zona superior. Ahora todos los eventos de ratón

Figura 2.3: Configuración del teclado

y teclado que produzcan sonidos se irán registrando hasta que pulsemos el botón *Detener*. Una vez pulsado este botón volveremos a ver que la zona de recogida de eventos ha vuelto a ser etiquetada como *Play Zone*, y cualquier nuevo sonido que hagamos no será almacenado. Si queremos oir la grabación que hemos hecho, podemos pulsar el botón *Reproducir*. Además, podemos silenciar las pistas que no queramos oir seleccionando las casillas etiquetadas con *Mute*. Si grabamos una nueva pista observaremos cómo se reproduce al mismo tiempo la pista que grabamos con anterioridad. Este mecanismo se denomina *full-duplex*, lo que quiere decir que se está reproduciendo sonido anteriormente grabado a la vez que se graban nuevas pistas. La implementación subyacente al *full-duplex* se detalla en la sección 6.5, página 33.

Siguiendo estos pasos podemos grabar hasta cuatro pistas diferentes. También podemos cambiar de instrumento pulsando en *Violin* y seleccionando el nuevo instrumento al que queramos asociar la pista en cuestión. Los distintos tipos de instrumentos de los que dispone PUMUKI se detallan en la sección 5.2.3, página 27. La reverberación de cada pista es configurable igualmente moviendo la barra de desplazamiento que hay en cada pista etiquetada como *Reverb*.

Si no estamos satisfechos con alguna pista grabada, podemos volver a seleccionar la casilla *Rec* de esa pista y volverla a grabar de nuevo pusando el botón *Grabar*, tantas veces como queramos. Una vez hayamos completado nuestra composición con éxito y estemos absolutamente satisfechos, podemos salvar nuestro proyecto, acción que se detallará en la siguiente sección.

2.4. Salvando y cargando un proyecto

Para salvar un proyecto pulsamos en *Archivo->Save Project* e introducimos en el panel emergente el nombre con el que quermos guardar nuestro proyecto. Esta acción guardará todas las características del proyecto que acabamos de crear, incluidos los instrumentos asociados a cada pista.

Si lo que queremos es cargar un proyecto anteriormente almacenado, pulsaremos *Archivo->Import Project* y seleccionaremos el archivo con extensión `pmk` del proyecto que queramos cargar.

2.5. Exportando un proyecto

PUMUKI también incluye otra poderosa funcionalidad: la posibilidad de exportar un proyecto abierto a formato WAV. Esta conversión se realiza de la siguiente manera: una vez que tenemos nuestro proyecto creado, o hayamos cargado el proyecto que queremos exportar, pulsamos el botón *Convertir* que se muestra a la derecha de la barra de botones. Esta acción generará automáticamente un archivo con nombre `testwav.wav`.

2.6. Uso del metrónomo

Tenemos la opción de grabar nuestras pistas con la ayuda de un metrónomo. El modo de uso es simple, cuando queramos usar un metrónomo símplemente marcamos el bóton de la interfaz gráfica principal que lleva este nombre (ver sección 2.2, página 18) y para controlar los *bpm*[1] o pulsos por minuto podemos usar la barra que está a la derecha de dicho botón. Cuando entremos en modo de grabación y estemos haciendo uso del metrónomo, éste se sincronizará precisamente en el momento en que pulsemos el botón *Record*.

[1] beats per minute

Capítulo 3

Interfaz gráfica

Antes de meternos con el diseño de la interfaz gráfica de PUMUKI vamos a ver qué alternativas planteamos y las ventajas e inconvenientes de cada una de ellas.

3.1. Opciones planteadas

Inicialmente se plantearon tres librerías gráficas posibles para implementar nuestra parte gráfica, a saber: QT, FLTK y GTK. En una primera selección decidimos suprimir las QT ya que no nacieron como *open source*, no obstante es la librería que, a día de la redacción, utiliza uno de los escritorios más populares de Linux como es KDE. Nos quedaron dos alternativas, cuyos pros y contras vemos en los siguientes puntos.

3.1.1. FLTK

Aporta compatibilidades OpenGL/Mesa; se desarrolla sobre las librerías gráficas de los sistemas operativos en los que se ejecuta, para aportar mayor velocidad y facilidad a la hora de una optimización de código y mejora del rendimiento; tiene compatibilidad a bajo nivel entre las plataformas gráficas sobre las que se sustenta (X11,Win32,MacOS) ya que sólo un 10 % es diferente en cada caso; aporta organización de las diferentes estructuras en una jerarquía de clases fácilmente entendible, si bien también se pueden usar las funciones C (*fl_functions*) de las que hacen uso las desarrolladas para C++; presenta problemas a los desarrolladores con algunas de sus estructuras (por ejemplo los FLTabs) que provocan mal funcionamiento de aplicaciones de complejidad media, aunque la realización de interfaces gráficas simples es bastante sencilla; tiene una pobre documentación en relación al tiempo de vida (la versión 1.1.4 data de 1998). Actualmente se encuentra en la versión 2.0.

3.1.2. GTK

Tiene una amplia aceptación por desarrolladores de software libre por ser la librería de la que hace uso GIMP; aporta programación orientada a objetos usando la librería GTKmm; es más compleja que FLTK para desarrolladores noveles aunque sus componenetes, hasta donde llega el conocimiento del redactor, no aportan problemas; tiene herramientas para desarrollar rápidamente las interfaces como por ejemplo *Glade*; presenta una fácil migración a otros sistemas operativos (como Windows) gracias a la existencia de herramientas que se encargan de hacer la conversión, en caso de que proceda; amplia documentación on-line [6] y en forma de libros. A la hora de redactar estas líneas GTKmm se encuentra en la versión 2.4

3.2. Estructura del interfaz

Al final se eligió *GTK* para implementar el interfaz gráfico, cabe destacar que impone una implementación tipo *modelo-vista-controlador*. Se basa en un sistema de cajas y una ventana principal. Intentaremos dar una pequeña aproximación a cómo estan organizadas esas cajas y distinguir qué parte es vista, que parte es controlador y que parte es modelo.

Antes de meternos a ver cómo está montada nuestra interfaz gráfica vamos a dar, a modo de ejemplo, una cabecera estandar de las funciones que servirán para nuestro controlador.

```
void genericFuntion(GtkWidget* widget, gpointer data)
```

Donde *widget* es la vista o componente que ha generado/detectado el suceso que tenemos que controlar y *data* es el modelo que vamos a usar en el controlador.

Partimos de una ventana principal, a la que se le da la posibilidad de capturar eventos de pulsación/despulsación de teclas cuyo uso inicialmente es saber que nota se esta pulsando en cada momento. La posibilidad de recoger estos eventos y su procesamiento es parte del controlador.

Ahora usando una caja principal, introduciremos todas las subsecciones a modo de sub-cajas y otros elementos que necesitamos. Dividimos esta caja principal en tres partes: una barra de menu, una barra de herramientas y otra caja (a partir de ahora VMainBox). La barra de menu contiene todos los menús/submenús de nuestra aplicación, cada uno de ellos controlados por sus controladores respectivos, pasa algo similar con la barra de herramientas.

En *VMainBox* necesitamos hacer alguna subdivisión más la dividimos en una caja superior (*VBoxTracks*) y otra inferior en la que metemos un *notebook* que será capaz de detectar eventos de ratón, tales como desplazamientos, pulsación de botones, ... cada uno asociado a una acción predeterminada.

Los controladores de los eventos de ratón y de teclado son los encargados de generar las instrucciones SKINI que correspondan en cada caso, valiéndose de componentes del evento capturado para generarlas.

Por último *VBoxTracks* se divide en cuatro subcajas idénticas que permiten controlar, en cierta medida, la pista que tienen asociada. Gracias a estas cajas podemos silenciar una pista, grabar en esa pista, fijar el instrumento de la misma y su reverberación y ver el tiempo relativo de la misma con respecto a la pista que más dura.

Capítulo 4

SKINI

SKINI[1] es un *pseudo-lenguaje* musical compuesto por mensajes de texto. Vamos a ver las características relevantes de dicho formato así como el significado de los mismos en las siguientes secciones.

4.1. Características principales

SKINI es compatible con MIDI en la medida de lo posible, esto es, es más rico que el formato MIDI. Partimos de la base de que SKINI son mensajes de texto así que cualquier sistema capaz de generar mensajes de texto puede generar SKINI, por analogía cualquier sistema capaz de procesar una cadena de texto puede procesar mensajes SKINI y que sea un formato basado en mensajes de texto da una gran facilidad de comprensión y capacidad de composición a los humanos. El cambio de MIDI a SKINI es válido, mientras que por norma general NO se puede pasar de SKINI a MIDI sin perder precisión ni riqueza del fichero que se se migra a MIDI.

4.2. Mensajes SKINI

Esta formado por una línea de texto plano, con tres campos que son obligatorios a saber: el tipo del mensaje que es una cadena de teto ASCII, el tiempo y el canal, esto es, un mensaje válido tiene que tener este formato:

```
mensaje tiempo canal
```

Para el tiempo tenemos tres valores e interpretaciones posibles: tiempo real, tiempo relativo y tiempo absoluto.

En versiones posteriores de SKINI (actualmente se encuentra en la 0.9) se podrán concatenar varios mensajes en una misma línea separados con

[1] Synthesis toolKit Instrument Network Interface

el operador *';'*. El campo `mensaje` tiene todos los tipos que incluye MIDI, tales como `NoteOn`, `NoteOff`, mensajes que extienden a MIDI, como `StringDamping` o `LipTension` e incluye otros mensajes que NO son MIDI como `SetPath`.

4.3. Parser de SKINI

Vamos a dar algunas aproximaciones para implementar correctamente un *parser* de SKINI:

- En un fichero SKINI podemos introducir comentarios, todas las líneas que comiencen por `/` serán tratadas como comentario y no tratadas.
- Los tabuladores, espacios y líneas en blanco no provocan ningún efecto.
- El campo `tiempo` debe representar un valor en segundos, que puede ser 0.0 para tratar con tiempo real, un valor que comience con = para indicar tiempo absoluto (por ejemplo `=0.2000`) y un valor decimal que indica tiempo relativo con respecto a la última instrucción.
- El campo `canal` debe ser un entero que puede tener cualquier valor que pueda tomar un `long int`, este hecho es una de las razones de no poder pasar un fichero SKINI a MIDI, ya que los canales MIDI están limitados a valores entre 0 y 15.
- Puede haber más campos en una instrucción SKINI pero son opcionales.

Capítulo 5

Generación del sonido

PUMUKI se apoya en las librerías STK[1] desarrolladas por Perry R. Cook y Gary P. Scavone, de la universidad de Stanford [3].

5.1. STK

El Synthesis Toolkit es un conjunto de clases de código libre para la síntesis algorítmica y el procesamiento de audio escritas en el lenguaje de programación C++. STK es multiplataforma, y permite la generación de audio en tiempo real para las plataformas SGI (Irix), Linux, Macintosh OS X, y Windows. Al ser código libre, STK tiene una gran extensibilidad por parte de los usuarios, ya que no incluye librerías no habituales o drivers *privados*.

La elección para nuestro proyecto de estas librerías vino determinada por su facilidad de uso, su gran cantidad de instrumentos implementados con la gran mayoría de métodos de síntesis, todos los cuales generan sonidos con una calidad más que aceptable, su portabilidad, y la potencia que otorga al ser usada en combinación con el lenguaje de control SKINI.

5.2. El proyecto *demo*

5.2.1. Introducción

El uso que hacemos en PUMUKI de estas librerías es a través de uno de los proyectos incluídos con las librerías. El proyecto *demo* incluído con STK permite comprobar el funcionamiento de todos los instrumentos que se incluyen en la librerías. Además, este proyecto permite la síntesis de sonido mediante instrucciones SKINI en *tiempo real*, capacidad que es explotada al máximo por PUMUKI. La comunicación entre el generador de instrucciones SKINI de PUMUKI y la síntesis de sonido STK se detalla en la sección 6.6,

[1] Synthesis Toolkit

página 33. Finalmente, el proyecto demo también incluye otra capacidad que PUMUKI explota al máximo, la polifonía.

El uso de estas funcionalidades se detalla a continuación.

5.2.2. El tiempo real en *demo*

El proyecto demo, además de otras muchas funcionalidades, nos permite la síntesis de sonido en tiempo real usando instrucciones de control SKINI. Para esto, se debe usar la opción `-or`. A partir del momento en que se ejecuta el proyecto demo con esta opción, las instrucciones SKINI que se envíen a la entrada estándar de demo serán procesadas inmediatamente y el sonido será sintetizado en tiempo real. Para esto, se deben usar instrucciones SKINI con tiempo delta igual a 0 (ver capítulo 4 en página 24 para más detalles sobre los distintos usos de los tiempos en las instrucciones SKINI).

5.2.3. Polifonía en *demo*

La función de polifonía en demo se puede invocar usando la opción `-n` *N* donde *N* es el número de canales máximo que pueda procesar demo. Para el caso de PUMUKI, necesitamos 5 canales, 4 de los cuales serán destinados a los instrumentos de los proyectos PUMUKI y el quinto será el canal donde el metrónomo que va marcando los tiempos, con lo cual la opción que usará PUMUKI será *-n 5*.

Inicialmente, el proyecto demo permite el uso del mismo instrumento para todos los canales de que disponga la aplicación. Es por esto que nos vimos obligados a incluir una nueva instrucción SKINI (`InstrmntChange`) que nos permitiese asignar a cada canal un instrumento distinto. Los detalles sobre la implementación de esta nueva instrucción se detallan en la sección 6.7, página 34.

5.2.4. Los instrumentos incluidos en *demo*

Los instrumentos que se pueden utilizar en PUMUKI, que son los disponibles en la versión 4.2.1 de STK son:

- **Clarinet**: Modelado físico del clarinete.
- **BlowHole**: Un modelo físico del clarinete con un agujero de tonalidad y un registro de viento.
- **Saxofony**: Un modelo físico mediante un instrumento pseudo-cónico que emula el sonido del saxofón.
- **Flute**: Un buen modelo físico de la flauta de pico.
- **Brass**: Modelado físico de un instrumento de viento.

- **BlowBotl**: Un resonador básico *helmholtz* que emula el sonido de una botella al ser soplada.
- **Bowed**: Modelado físico de un instrumento de cuerda que intenta representar el sonido de un violín.
- **Plucked**: Un modelo físico básico de una cuerda pulsada.
- **StifKarp**: Modelado de Karplus-Strong de una cuerda pulsada.
- **Sitar**: Modelado físico de una cuerda que emula el sonido del sitar.
- **Mandolin**: Modelado físico de la mandolina de dos cuerdas.
- **Rhodey**: Modelado de síntesis FM del piano eléctrico tipo Rhodes.
- **Wurley**: Modelado de síntesis FM del piano eléctrico modelo Wurlitzer.
- **TubeBell**: Modelo de síntesis FM de una campana.
- **HevyMetl**: Modelo de síntesis FM de un sintetizador distorsionado.
- **PercFlute**: Modelo de síntesis FM de una flauta percusiva.
- **BeeThree**: Modelo de síntesis FM de un órgano.
- **Moog**: Sampler orientado a filtros de frecuencia.
- **FMVoices**: Síntesis de tres voces FM
- **VoicForm**: Síntesis de voz a través de un filtro de resonancia formado por cuatro voces.
- **Resonate**: Ruido a través de un filtro *BiQuad*.
- **Drummer**: Síntesis de sampleado de una batería.
- **BandedWG**: Tabla de ondas que permite la generación de sonidos como cuencos tibetanos, etc.
- **Shakers**: Varios modelos estocásticos de eventos que modelan instrumentos tipo maracas.
- **ModalBar**: Varias preconfiguraciones de cuatro resonancias (marimba, vibráfono, etc.).
- **Mesh2D**: Tabla de ondas digital dos dimensiones.
- **Whistle**: Modelado híbrido físico/espectral de una sirena de policía.

5.2.5. Funcionamiento

A grandes rasgos el funcionamiento es el siguiente, el programa va leyendo de la entrada estandar las instrucciones SKINI que le llegan, realiza un análisis de las mismas y ejecuta las acciones pertinentes a las mismas. Para realizar este proceso se hace uso de la siguiente estructura

```
struct TickData {
WvOut **wvout;
Instrmnt **instrument;
Voicer *voicer;
Effect *reverb;
Messager messager;
Skini::Message message;
StkFloat volume;
StkFloat t60;
unsigned int nWvOuts;
int nVoices;
int currentVoice;
int channels;
int counter;
bool realtime;
bool settling;
bool haveMessage;
}
```

El funcionamiento del reproductor de STK es el siguiente, se crea un hilo para procesar el sonido, éste sigue el formato *callback* de STK, esto es, la función tick *ver programa en ejecución* es llamada por el hardware de audio del sistema cuando éste se ve capacitado para aceptar más datos. Cada vez que el sistema llame a dicha función se procesan todos los mensajes existentes en la estructura anterior. El procesamiento del mensaje simplemente consiste en un parseo de cada string y ejecutar las acciones pertinentes en relación a dicha instrucción.

El atributo voicer es el encargado de controlar la polifonía, controla conjuntos de instrumentos asociando éstos a canales y agrupándolos por medio de la función *addInstrument*. La función *tick* devuelve la mezcla de todas las voces que están sonando en un momento determinado. Además es el encargado de procesar los mensajes y ejecutar, por medio de funciones miembro, los mensajes SKINI que recibe la estructura.

Para la comunicación multicanal entre la aplicación y el hardware del sistema se usa el atributo WvOut escribiendo un argumento en todos los canales, ahora bien, si estamos en tiempo real el numero de canales que se puede soportar depende del hardware del sistema(para nuestros equipos sólo se soportan dos canales).

Capítulo 6

Implementación de PUMUKI

6.1. Introducción

Al hacer uso de GTK como herramienta de implementación el programa principal lanza un hilo para gestionar la interfaz gráfica. Ésto se hace a través del método `gtk_main()` que está encargado del refresco de la GUI. El programa principal espera este hilo, esto es, para que siga ejecutando las instrucciones que estén por debajo de esta función se debe cerrar la ventana principal de nuestra aplicación. Una vez estemos dentro de la función `gtk_main()` nuestra interfaz gráfica sólo responde a interrupciones o bien generadas por eventos que captura, o bien generadas porque nosotros fijemos que cada cierto tiempo se ejecute una función determinada. A continuación vamos a dar una introducción a las funciones más interesantes que hemos tenido que implementar.

6.2. Generación de instrucciones SKINI

La generación de instrucciones SKINI tiene su origen en la interactuación de *PUMUKI* con los usuarios.Éstos generan eventos cuando se pulsa una tecla o se mueve el ratón. Vamos a ver el *¿qué?* y el emph¿cómo? de cada uno de ellos, además de intentar dar una idea de los problemas que se presentaron en el tratamiento de los mismos.

6.2.1. Los eventos del ratón

Aquí podemos distinguir dos acciones, a saber: los eventos generados porque el puntero del ratón ha cambiado de posición y los provocados por las diversas pulsaciones de los diferentes botones de nuestro periférico.

Eventos de movimiento

Cada vez que se mueve el ratón se genera un evento indicando los nuevos valores de coordenadas de la posición a la que hemos llevado el puntero. Para nuestro sistema ésto, que en una primera aproximación parece que es lo deseable, nos ocasionó un problema. La razón es clara, no interesa que se generen las instrucciones asociadas siempre que ocurra uno de estos eventos, ya que cualquier desplazamiento del ratón genera un gran número de ellos (uno por cada cambio en cada coordenada del puntero), así que una de las soluciones es guardar los cambios de coordenadas y cada cierto tiempo mirar cual es la diferenciaentre la posición en la anterior observación y la posición actual, considerando el valor para la generación de la instrucción asociada la diferencia de estas. El tiempo entre observaciones es un parámetro importante, si las observaciones se realizan en intervalos muy pequeños el efecto conseguido se aproximará al caso en el que no se realice ninguna observación mientras que si se espacian mucho en el tiempo la diferencia entre los puntos puede ser muy grande. Después de diferentes pruebas con distintos valores, fijamos un periodo de tiempo de *100 milisegundos* llegando a un compromiso entre latencia y valores obtenidos entre las observaciones. La primera idea para generar cambios con este parámetro es el cambio de sonido considerando la diferencia de módulos de los puntos actual y anterior.

Uno de los problemas que surgen en la generación de sonido con este método es la siguiente, si se producen variaciones muy bruscas y distantes de las posiciones anterior y actual del puntero del ratón se genera un crujido. Ésto es debido a cambios de volumen muy bruscos en un corto periodo de tiempo, la solución a esto es meter un filtro de un polo que limite el rango de variación del volumen y no se generen dichos crujidos.

Eventos de pulsación

Actualmente se cogen las pulsaciones del botón izquierdo del ratón y del botón derecho, sin embargo no genera ninguna instrucción, queda pendiente para futuras versiones.

6.2.2. Los eventos del teclado

Cada vez que se pulsa una tecla, PUMUKI captura ese evento y lo gestiona, supondremos que el teclado que tenemos es un teclado *QWERTY*. Dividimos el teclado en dos secciones, la primera contiene la fila de la 'Z' y la fila de la 'A', y la segunda contiene la fila de la 'Q' y la fila del '1',en cada una de las secciones mapeamos las teclas como si se tratase de un teclado de piano, considerando la letra 'Z' (respectivamente la 'A') como el DO. La sección superior está mapeada una escala más aguda que la sección inferior. Para subir/bajar de escala se pueden usar las teclas *F4*/*F3* respectivamente. La escala inicial de la primera sección es la quinta. Cada vez que se pulsa

una tecla recogemos ese evento y buscamos la nota asociada a la misma, si estaba sonando alguna y esta nota es más aguda, hacemos que suene esta, sino la marcamos como pulsada. Cuando soltamos una tecla, la desmarcamos como pulsada y hacemos que suene la siguiente nota más aguda que esté pulsada, caso de existir.

6.3. Cambios de volumen y escalas

Para generar cambios en el volumen de los instrumentos se considera la diferencia de coordenadas del ratón entre la posición actual del ratón y la posición donde estaba en el último chequeo. Con esta diferencia de coordenadas se llama al generador de instrucciones, el cual, dependiendo del tipo de escala que tenga como *escala actual* multiplicará por un determinado factor el volumen que ha recibido. Establecimos por medio de ensayo y error los siguientes factores:

- Para una escala actual tipo logarítmico

```
newVolume = 45*log(difference)
```

- Para una escala de tipo lineal

```
newVolume = 0.06*difference
```

Siendo `difference` el parámetro que recibe la función que genera la instrucción SKINI de los cambios de sonido y `newVolume` el parámetro de la instrucción SKINI que va a llevar el cambio de volumen.

6.4. Mezclando las pistas

Cada vez que se acaba de grabar una pista se lleva a cabo un proceso de mezclado de todas las pistas que tenemos grabadas en nuestro proyecto. Cuando grabamos una pista, se va grabando la instrucción con el tiempo relativo a la anterior como parámetro de tiempo de la instrucción anterior, para ello llevamos un contador del tiempo total que llevamos grabando. A continuación tenemos que mezclar todas las pistas anteriormente grabadas (que están contenidas en el fichero `merge.ski`) con la pista que acabamos de grabar. Se leen las instrucciones de ambos ficheros, se considera la de tiempo menor y se escribe en el fichero de salida, a continuación se resta el tiempo de la instrucción escrita a la otra instrucción, y se lee una nueva instrucción del fichero al que pertenecía la instrucción escrita. Este proceso se itera hasta que se acaba uno de los dos ficheros, momento en el que se escribe el resto de instrucciones en el fichero de salida.

6.5. Reproduciendo un fichero

El fichero que tiene todas las pistas de nuestro proyecto va a ser reproducido en dos situaciones cuando pulsemos el botón de play(por razones evidentes) y cuando pulsemos el botón de *record*, en este último caso se plantea un problema productor-consumidor. Tenemos dos hilos que tienen que generar instrucciones para el mismo consumidor(el reproductor de sonido en GTK), tenemos que intercalar las instrucciones del archivo con las instrucciones que generan los eventos del usuario. Las instrucciones del usuario no deben tener latencia por ser tiempo real, mientras que las instrucciones del fichero tienen que reproducirse en los mismos momentos en los que fueron grabados con el fin de no tergiversar la grabación. Hay que ver cómo se intercalan las instrucciones de ambos lugares para evitar que solo se consuman las instrucciones de uno de los productores o bien que no se atiendan las mismas cuando corresponda. La solución que se adoptó fue la siguiente, STK es capaz de reproducir varias instrucciones que con los parametros de tiempo real, así que cada vez que se va a reproducir un fichero se realizan los siguientes pasos:se lee la instrucción correspondiente del fichero, se toma el tiempo relativo de la instrucción, se espera ese tiempo para lanzar la instrucción y si corresponde a una pista que no está silenciada o a la pista en la que estamos grabando se lanza con formato de instrucción de tiempo real.

6.6. La comunicación del generador de instrucciones SKINI con STK

Como se ha explicado con anterioridad, PUMUKI procesa los eventos de teclado y ratón que el usuario haga para transformarlos en instrucciones SKINI y enviarlos a las librerías de síntesis STK. Pero, ¿cómo se lleva a cabo esta comunicación entre PUMUKI y STK?.

Para resolver este problema, valoramos varias maneras de solucionarlo:

- Integrar de manera definitiva las librerías STK en nuestra aplicación PUMUKI.
- Utilizar algún tipo de mecanismo de paralelización de aplicaciones (OpenMP o pthreads) para lanzar simultáneamente un proceso productor de instrucciones y otro consumidor (STK).
- Usar una tubería de comunicación entre procesos (*pipe*) para redireccionar la salida estándar del generador de instrucciones a la entrada estándar de las librerías STK.

Después de analizar las principales opciones con detenimiento, nos decantamos por la última opción. La posibilidad de integrar las librerías STK

en PUMUKI, a pesar de ser la que en principio nos atrajo más, finalmente se presentó como un problema de gran envergadura para el tiempo de que disponíamos, principalmente por la enorme complejidad de estas librerías, además de poder disponer de otras soluciones no tan elegantes como esta pero igual de efectivas.

Por otro lado, la opción de incluir paralelismo en la aplicación, aunque en principio tomó bastante fuerza, finalmente también fue descartada porque al tratarse de una aplicación que trabaja en tiempo real no permitiría ningún tipo de retrasos que el uso de estos mecanismos de paralelización pudiesen acarrear.

La posibilidad de usar un pipe como medio de comunicación entre estos dos procesos finalmente fue la seleccionada debido a su gran sencillez y poderosa efectividad. El hecho de implementarlo de esta manera además nos ha permitido llevar un proceso de desarrollo más modularizado, al ser independientes completamente el módulo de producción de instrucciones SKINI y el módulo consumidor de instrucciones (STK).

6.7. Cambiando de instrumento

Para poder cambiar de instrumento durante el uso de nuestra aplicación tuvimos que efectuar algunos cambios en el proyecto demo. Como se indica en el capítulo 5, inicialmente el proyecto demo incluye la instrucción `ProgramChange`, que permite cambiar los instrumentos asociados a todos los canales. El inconveniente de esta instrucción es que no nos permitía asignar a cada canal un instrumento diferente, lo cual era un objetivo principal de nuestra aplicación.

Para poder incluir esta funcionalidad, tuvimos que modificar las librerías de síntesis STK, en particular el proyecto demo, para que incluyese una nueva instrucción que nos permitiese modificar individualmente el instrumento asociado a cada canal. Esta instrucción fue denominada `InstrmntChange` y tiene cuatro parámetros que indican el instrumento asociado a cada uno de los canales. Por ejemplo, la instrucción

```
InstrmntChange Bowed Clarinet Plucked Rhodey
```

asocia al canal 1 el instrumento tipo violín, al 2 un instrumento tipo clarinete, al 3 un instrumento tipo guitarra, y al canal 4 un instrumento tipo piano *Rhodes*.

La instrucción InstrmntChange nos ha obligado a incluir en los ficheros SKINI.msg, SKINI.msg, Skini.h, Skini.cpp y demo.cpp de las librerías STK la información necesaria para que el parser de SKINI recogiese las instrucciones de este tipo y las procesase. Básicamente, hemos tenido que incluir un nuevo campo `_tokens`en la estructura de tipo Message (que se incluye en Skini.h) que será un vector de strings que incluirá los cuatro instrumentos

a los que van a estar asociados los cuatro canales. Además, en el `demo.cpp` debemos incluir un nuevo caso en el procedimiento `proccessMessage` para que cuando llegue una nueva instrucción de tipo InstrmntChange se cambien los instrumentos asociados a cada canal.

El listado de estas modificaciones se incluye en el apéndice B, página 46.

6.8. Implementación del metrónomo

Uno de los problemas que surgen cuando se quiere realizar una tarea periódica reside en el modo de hacerla. Si el periodo de la misma es siempre constante símplemente fijamos los parámetros y lanzamos un hilo que ejecute dicha tarea. Ésto en nuestro caso no vale por diversos motivos que veremos a lo largo de esta sección el periodo de la tarea que queremos realizar, ejecutar un tono cada cierto tiempo, no es constante ya que depende del valor que quiera darle el usuario, además la solución del hilo no parecía atractiva a priori, teníamos que preocuparnos de varios factores, como por ejemplo el que estuviese el metrónomo activo y el valor de los tonos por segundo; además habría que meter un temporizador en la función para que mandase al módulo de sonido el tick cuando acabase.

GTK nos dio una solución que nos pareció más atractiva para la implementación de esta funcionalidad, se permite fijar tareas que se ejecutan en un periodo variable de tiempo por medio de temporizadores identificados por un *id* usando la función `g_timeout_add(timeout,function,data)` que recibe el valor del periodo de dicha tarea en milisegundos, la tarea, y los datos que necesita la función. Ahora bien, no es nada interesante ir acumulando temporizadores sobrecarga el sistema además de desvirtuar la funcionalidad del metrónomo así que cuando cada vez que se varían los tonos por segundo se elimina el temporizador actual y se crea otro con el nuevo valor.

Se garantiza que hay sólo un temporizador correspondiente al metrónomo. La función que genera el sonido del metrónomo es bastante simple, se genera una instrucción skini sobre el canal 5 que tiene prefijado el tipo de sonido que queremos para el metrónomo. En nuestro caso actual dicha instrucción es:

```
NoteOn 0.0 5 56 127.00
```

y el sonido que está fijado en el canal es el correspondiente al instrumento `Drummer`.

Capítulo 7

Gestión del proyecto

7.1. Introducción

Una de las partes fundamentales en un proyecto de cierta envergadura es su gestión y la comunicación entre los componentes del equipo de desarrollo. En este capítulo intentaremos dar una visión de nuestro formato de comunicación y modos de la misma, además de ver como hemos enfocado el desarrollo del proyecto.

7.2. Modelo de proceso y mantenimiento del código

Se ha seguido un modelo de proceso en espiral con generación de prototipos al final de cada ciclo. Intentamos resolver los siguientes problemas:

- *Requisitos del sistema*: teníamos que determinar cómo ibamos a obtener e interpretar los eventos que debían ser tenidos en cuenta como las pulsaciones de teclas y el movimiento del ratón.
- *Objetivos*: el resultado final que deseamos era una aplicación capaz de obtener cierta capacidad de interpretación en la generación de música por medio de unos periféricos muy básicos, y poder obtener composiciones polifónicas de cierta calidad.
- *Diseño lógico*: esta parte es bastante importante a la hora de realizar un proyecto, si conseguíamos tener un buen diseño tendríamos un código fácilmente mantenible. Aquí usamos ciertas ideas de patrones de diseño como el patrón *fachada* y el patrón *modelo-vista-controlador*

La forma de mantenimiento del código que usamos no fue la más ortodoxa ya que no usamos un sistema de control de versiones ya implementado como pueden ser *CVS o SVN*, ahora bien, aplicamos las ideas subyacentes a éstos

sistemas. Simplemente cuando alguien realizaba modificaciones en algún fichero se hacía una comparación con la última versión (la cual se transmitía vía correo electrónico) se resolvían las diferencias con ayuda de *Kdiff3*, se volvía a chequear si había una nueva versión y se enviaba el resultado.

7.3. Reuniones y asignación de tareas

En todo equipo de desarrollo la comunicación entre miembros es crucial, de hecho una falta o deficiencia en los modos de comunicación puede condenar al fracaso el proyecto, a fin de paliar todos los efectos negativos de una mala comunicación, se ha mantenido un contacto continuo y periódico entre los integrantes del equipo de desarrollo con reuniones cara a cara semanales y reuniones con el director de proyecto cuando los avances realizados tenían cierta relevancia y envergadura, normalmente cada dos semanas.

Cada miembro del equipo escogía sus tareas a realizar según su disponibilidad y conocimientos en las reuniones, además de encargarse de depurar y testear el código que generaba.

Capítulo 8

Estado del arte

Vamos a dar una ligera introducción a los programas que hacen uso del modelado físico para generar sonido.

8.1. Vir Syn Tera 2

Es una workstation virtual que una gran cantidad de formas posibles de generar sonido, tales como: waveshaping, síntesis análoga, síntesis espectral, síntesis FM y modelado físico, entre otros. Entre otras funcionalidades permite al usuario obtener sonidos al azar y microafinación. La interfaz que usa modelado físico recibe el nombre de *waveTERA* la cual, además de simular sonidos reales, permite modificar los parámetros del modelado para obtener sonidos que, en principio, serían imposibles de obtener en la vida real.

8.2. Chuck

Se trata de un lenguaje para tiempo real multiplataforma, originalmente desarrollado por *Ge Wang* y *Perry Cook* con un nivel de expresividad fundamental, con la característica de poder modificar código al vuelo mientras el programa está ejecutándose. Soporta sonido multicanal, MIDI, OSC y HID.

8.3. Calico

Es una aplicación en línea de comandos que permite crear y grabar sonidos usando los instrumentos de STK, sintetizadores MIDI, o samples usando partituras que están contenidas en ficheros que contienen listas de eventos musicales. Es una herramienta muy válida para generar música a través de lenguajes de scripts.

8.4. MIC Modeler, los micrófonos virtuales de Antares

La empresa Antares desarrolló una tecnología que permitió construir modelos digitales de las características de sonido de los micrófonos más conocidos. Éstos modelos permiten obtener las particularidades de cada micrófono. El modo de uso es simple, simplemente se selecciona la pista en la que se quiere realizar la grabación, elegimos el micrófono modelado y A GRABAR!

8.5. TAO

Tao es una aplicación software para síntesis de sonido usando modelado físico. Proporciona instrumentos virtuales implementados, usando el modelo de masas, a partir de los cuales se pueden crear instrumentos más complejos. Sus principales características son:

- Posibilidad de crear instrumentos complejos a partir de sus instrumentos virtuales.
- Animación en OpenGL de la propagación de las ondas a través del instrumento.
- Posibilidad de obtener ficheros wav.
- Buena documentación
- Estar bajo licencia GPL.

8.6. PureData

Es un entorno multiplataforma en tiempo real para el procesamiento de gráficos, vídeo y audio cuyo núcleo actualmente mantenido por Miller Puckette. Está basado en la familia de lenguajes de programación conocidos como Max. Surge para ver cuánto se puede mejorar el paradigma Max. Contiene una librería de objetos que están implementados usando modelado físico y otra para generar y procesar vídeo en tiempo real.

Capítulo 9

Conclusiones

Vamos a intentar dar una idea general de la evolución del proyecto desde los objetivos establecidos inicialmente hasta posibles ampliaciones del proyecto.

9.1. Revisión de objetivos iniciales

Uno de los principales objetivos del proyecto era crear una herramienta de composición musical a través del uso de modelado físico, posibilitando la interpretación. Para la inclusión de la interpretación se pretendía usar un lenguaje de script, además del uso de los periféricos para la composición. Inicialmente sólo se dispondrá de monofonía.

Esta parte de los objetivos se ha conseguido satisfactoriamente, la aplicación es capaz de dar cierta capacidad de interpretación usando el ratón para variar el volumen de la nota y el teclado para la generación del sonido. Todos estos eventos se traducen a un lenguaje de script como es SKINI y el sonido corresponde a un único instrumento fijo, el violín.

Se intenta realizar un secuenciador basado en todo lo desarrollado anteriormente con la capacidad de mezclar pistas, introducir sonidos de varios instrumentos, permitir reproducción simultanea de varias pistas, además de tener en cada pista un instrumento diferente asociado. Todo esto se ha conseguido, se ha tenido que extender SKINI para poder cambiar los instrumentos asociados a cada pista en tiempo real, se puede generar WAV de la composición.

9.2. Ampliaciones posibles

Considerar interpretaciones diferentes para la variación de los parámetros dependiendo de los instrumentos, inclusión de más parámetros de entorno, posibilidad de cambiar el sonido del metrónomo en tiempo real, exportar las composiciones a formato MIDI. Dar un significado a la variación de la

rueda del ratón, poder cambiar el siginficado de la variacion de cada instrumento en tiempo real.

9.3. Evolución del proyecto

Inicialmente la cantidad de ideas que se tenían para la realización del proyecto eran abrumadoras, había varias opciones de desarrollo como ampliar programas ya existentes tales como TAO o PureData o bien realizar una nueva aplicación usando alguna librería ya existente, como al final se hizo. Dentro de estas opciones las posibles líneas de desarrollo son muy diferentes a la par que amplias. Debido a la complejidad inicial que supone el coger código ajeno e insertar nueva funcionalidad se decide partir *"desde cero"* con base en STK. Esta opción acotaba un poco las líneas de desarrollo, y nosotros fijamos unos parámetros que nos han acompañado a lo largo de este viaje, enfocamos el proyecto a realizar un secuenciador de sonido con capacidad interpretativa.

Una vez centradas las ideas iniciales, teníamos que realizar un estudio exhaustivo de la tecnología que ibamos a usar, hicimos una valoración desde la interfaz gráfica a las capacidades que nos ofrecía STK, pasando por el total entendimiento del lenguaje de script que es SKINI. Teníamos que comprobar si STK aceptaba el cambio de parámetros en tiempo real y una vez investigado ésto, si SKINI nos daba la posibilidad de realizar estos cambios. Realizando pruebas, surgió uno de los problemas inherentes al escritorio KDE de Linux debido a la gestión de sonido del mismo(*arts*). El sistema de sonido puede ser bloqueado para que el sistema lo use durante un intervalo de tiempo, el mismo problema se puede experimentar si se tienen programas viejos que también pueden apropiarse del sistema de sonido(como *XMMS*). Por ésto se decidió que tendríamos que hacerlo sobre GNOME, aunque ya sabíamos que era mucho más potente que KDE ;-)

Una vez cerrados éstos puntos surgieron más problemas, teníamos que decidir cómo capturar los eventos si bien meter rutinas en ensamblador o usar algún otro método. La solución vino dada por GTK, que es capaz de capturar los eventos lanzados por las X11 y nos ahorró el desarrollo tedioso del código en ensamblador.

Después de ésto teníamos que mapear el teclado, dar la funcionalidad al ratón... y a continuación llegó el momento en el que tuvimos el secuenciador monofónico operativo. A partir de aquí tuvimos que considerar otras muchas opciones, si bien podíamos ampliar la interpretación posible sobre nuestro instrumento consideramos que era mejor introducir polifonía para dar más riqueza musical. Horas de trabajo consiguieron que pudiesemos obtener una reproducción multicanal bastante aceptable pero para un sólo instrumento. Ahora bien, debido a la forma de comunicación con el módulo de sonido (vía pipe) teníamos que pensar en un modo rápido de obtener los cambios

y aquí llegó la idea de ampliar SKINI con una instrucción de cambios de instrumento para que pudiese implementar esta función.

Resumiendo al final tenemos un secuenciador que facilita la creación de piezas musicales de cierta calidad, que es sencilla para los amateurs y otorga cierta calidad y potencia para los usuarios avanzados.

Capítulo 10

Apéndice A: Instrucciones SKINI

Las instrucciones que conforman SKINI son las siguientes donde los campos son los opcionales,recuérdese que toda instrucción SKINI necesita tres campos obligatorios,ver 4.2:

- NoteOff, SK_NoteOff, SK_DBL, SK_DBL
- NoteOn , SK_NoteOn, SK_DBL, SK_DBL
- PolyPressure, SK_PolyPressure, SK_DBL, SK_DBL
- ControlChange, SK_ControlChange, SK_INT, SK_DBL
- ProgramChange, SK_ProgramChange, SK_DBL, SK_DBL
- AfterTouch, SK_AfterTouch, SK_DBL, NOPE
- ChannelPressure, SK_ChannelPressure, SK_DBL, NOPE
- PitchWheel, SK_PitchWheel, SK_DBL, NOPE
- PitchBend, SK_PitchBend, SK_DBL, NOPE
- Clock, SK_Clock, NOPE, NOPE
- Undefined, 249, NOPE, NOPE
- SingStart, SK_SongStart, NOPE, NOPE
- Continue, SK_Continue, NOPE, NOPE
- SongStop, SK_SongStop, NOPE, NOPE
- Undefined, 253, NOPE, NOPE

- ActiveSensing, SK_ActiveSensing, NOPE, NOPE
- SystemReset, SK_SystemReset, NOPE, NOPE
- Volume, SK_ControlChange, SK_Volume, SK_DBL
- ModWheel, SK_ControlChange, SK_ModWheel, SK_DBL
- Modulation, SK_ControlChange, SK_Modulation, SK_DBL
- Breath, SK_ControlChange, SK_Breath, SK_DBL
- FootControl, SK_ControlChange, SK_FootControl, SK_DBL
- Portamento, SK_ControlChange, SK_Portamento, SK_DBL
- Balance, SK_ControlChange, SK_Balance, SK_DBL
- Pan, SK_ControlChange, SK_Pan, SK_DBL
- Sustain, SK_ControlChange, SK_Sustain, SK_DBL
- Damper, SK_ControlChange, SK_Damper, SK_DBL
- Expression, SK_ControlChange, SK_Expression, SK_DBL
- NoiseLevel, SK_ControlChange, SK_NoiseLevel, SK_DBL
- PickPosition, SK_ControlChange, SK_PickPosition, SK_DBL
- StringDamping, SK_ControlChange, SK_StringDamping, SK_DBL
- StringDetune, SK_ControlChange, SK_StringDetune, SK_DBL
- BodySize, SK_ControlChange, SK_BodySize, SK_DBL
- BowPressure, SK_ControlChange, SK_BowPressure, SK_DBL
- BowPosition, SK_ControlChange, SK_BowPosition, SK_DBL
- BowBeta, SK_ControlChange, SK_BowBeta, SK_DBL
- ReedStiffness, SK_ControlChange, SK_ReedStiffness, SK_DBL
- ReedRestPos, SK_ControlChange, SK_ReedRestPos, SK_DBL
- FluteEmbouchure, SK_ControlChange, SK_FluteEmbouchure, SK_DBL
- LipTension, SK_ControlChange, SK_LipTension, SK_DBL
- StrikePosition, SK_ControlChange, SK_StrikePosition, SK_DBL

- StickHardness, SK_ControlChange, SK_StickHardness, SK_DBL
- TrillDepth, SK_ControlChange, SK_TrillDepth, SK_DBL
- TrillSpeed, SK_ControlChange, SK_TrillSpeed, SK_DBL
- Strumming, SK_ControlChange, SK_Strumming, 127
- NotStrumming, SK_ControlChange, SK_Strumming, 0
- PlayerSkill, SK_ControlChange, SK_PlayerSkill, SK_DBL
- Chord, SK_Chord, SK_DBL, SK_STR
- ChordOff, SK_ChordOff, SK_DBL, NOPE
- OpenFile, 256, SK_STR, NOPE
- SetPath, 257, SK_STR, NOPE

Donde los significados son los siguientes:

- NOPE: Indica que el campo no se usa, además indica que no habrá más campos en esta línea.
- SK_INT: byte, actualmente se parsea como un entero sin signo de 32 bits, si se necesita un campo MIDI que sea un entero se hace uso de este tipo de datos.
- SK_DBL: real de doble precisión en punto flotante. SKINI usa este tipo de datos en contextos MIDI para valores de notas, velocidades, valores de control,etc.
- SK_STR: sólo es válido si es el último atributo. Permite la inclusión de mensajes arbitrarios los cuales se escanean hasta encontrar un final de línea, por ejemplo, MIDI SYSEX(system exclusive) mensajes de 256 bytes puede ser leído como enteros delimitados por espacios dentro del buffer de 1k SK_STR.

Capítulo 11

Apéndice B: Listados de código

```
// demo.cpp
//
// An example STK program
that allows voice playback
and control of
// most of the STK instruments.

#include "SKINI.msg"
#include "WvOut.h"
#include "Instrmnt.h"
#include "PRCRev.h"
#include "Voicer.h"
#include "Skini.h"

#if defined(__STK_REALTIME__)
  #include "Mutex.h"
#endif

// Miscellaneous command-line
parsing and instrument allocation
// functions are defined in
utilities.cpp ... specific to this program.
#include ütilities.h"

#include <signal.h>
#include <iostream>
#include <algorithm>
```

```
#if !defined(__OS_WINDOWS__) // Windoze bogosity for VC++ 6.0
  using std::min;
#endif

bool done;
static void finish(int ignore){ done = true; }

// The TickData structure holds
all the class instances and data that
// are shared by the various processing functions.
struct TickData {
  WvOut **wvout;
  Instrmnt **instrument;
  Voicer *voicer;
  Effect *reverb;
  Messager messager;
  Skini::Message message;
  StkFloat volume;
  StkFloat t60;
  unsigned int nWvOuts;
  int nVoices;
  int currentVoice;
  int channels;
  int counter;
  bool realtime;
  bool settling;
  bool haveMessage;

  // Default constructor.
  TickData()
    : wvout(0), instrument(0),
 voicer(0), reverb(0), volume(1.0), t60(1.0),
      nWvOuts(0), nVoices(1), currentVoice(0),
 channels(2), counter(0),
      realtime( false ),
settling( false ), haveMessage( false ) {}
};

#define DELTA_CONTROL_TICKS 64 //
default sample frames between control input checks

// The processMessage() function encapsulates the
handling of control
```

```
// messages.  It can be easily relocated
within a program structure
// depending on the desired scheduling scheme.
void processMessage( TickData* data )
{
  register StkFloat value1 = data->message.floatValues[0];
  register StkFloat value2 = data->message.floatValues[1];
  std::vector<std::string> instrumentsList;
  int instruments;

  switch( data->message.type ) {

  case __SK_Exit_:
    if ( data->settling == false ) goto settle;
    done = true;
    return;

  case __SK_NoteOn_:
    if ( value2 == 0.0 ) // velocity is zero ... really a NoteOff
      data->voicer->noteOff( value1,
 64.0, data->message.channel );
    else // a NoteOn
      data->voicer->noteOn( value1, value2,
 data->message.channel );
    break;

  case __SK_NoteOff_:
    data->voicer->noteOff( value1, value2,
 data->message.channel );
    break;

  case __SK_ControlChange_:
    if (value1 == 44.0)
      data->reverb->setEffectMix(value2  * ONE_OVER_128);
    else if (value1 == 7.0)
      data->volume = value2 * ONE_OVER_128;
    else if (value1 == 49.0)
      data->voicer->setFrequency( value2,
 data->message.channel );
    else
      data->voicer->controlChange( (int) value1,
 value2, data->message.channel );
    break;
```

```
  case __SK_AfterTouch_:
    data->voicer->controlChange( 128, value1,
 data->message.channel );
    break;

  case __SK_PitchChange_:
    data->voicer->setFrequency( value1,
 data->message.channel );
    break;

  case __SK_PitchBend_:
    data->voicer->pitchBend( value1, data->message.channel );
    break;

  case __SK_Volume_:
    data->volume = value1 * ONE_OVER_128;
    break;

  case __SK_InstrmntChange_:
        instrumentsList = data->message._tokens;
    for ( instruments=0;
 instruments<data->nVoices; instruments++ )
      {
        //eliminamos el instrumento de cada canal
                data->voicer->
removeInstrument(data->instrument[instruments]);
                delete data->instrument[instruments];
      }

    for(instruments = 0;
 instruments<data->nVoices; instruments++)
      {
                //añadimos el instrumento que
tenemos en el vector de tokens
                data->currentVoice
= voiceByName((char*)instrumentsList[instruments].c_str(),
 &data->instrument[instruments]);
                if(data->currentVoice < 0)
                  data->currentVoice =
 voiceByNumber(0, &data->instrument[instruments]);
                data->voicer->addInstrument(data->
instrument[instruments],instruments+1);
                data->settling = false;
      }
```

```
        break;

  case __SK_ProgramChange_:
    if ( data->currentVoice == (int) value1 ) break;

    // Two-stage program change process.
    if ( data->settling == false ) goto settle;

    // Stage 2: delete and reallocate new voice(s)
    for ( int i=0; i<data->nVoices; i++ ) {
      data->voicer->removeInstrument( data->instrument[i] );
      delete data->instrument[i];
      data->currentVoice =
 voiceByNumber( (int)value1, &data->instrument[i] );
      if ( data->currentVoice < 0 )
        data->currentVoice =
 voiceByNumber( 0, &data->instrument[i] );
      data->voicer->addInstrument( data->instrument[i],
 data->message.channel );
      data->settling = false;
    }

  } // end of switch

  data->haveMessage = false;
  return;

 settle:
  // Exit and program change messages
are preceeded with a short settling period.
  data->voicer->silence();
  data->counter = (int) (0.3 *
data->t60 * Stk::sampleRate());
  data->settling = true;
}


// The tick() function handles
sample computation and scheduling of
// control updates.  If doing realtime
audio output, it will be called
// automatically when the system needs
a new buffer of audio samples.
int tick(char *buffer, int bufferSize, void *dataPointer)
```

```
{
  TickData *data = (TickData *) dataPointer;
  register StkFloat sample, *samples = (StkFloat *) buffer;
  int counter, nTicks = bufferSize;

  while ( nTicks > 0 && !done ) {

    if ( !data->haveMessage ) {
      data->messager.popMessage( data->message );
      if ( data->message.type > 0 ) {
        data->counter =
 (long) (data->message.time * Stk::sampleRate());
        data->haveMessage = true;
      }
      else
        data->counter = DELTA_CONTROL_TICKS;
    }

    counter = min( nTicks, data->counter );
    data->counter -= counter;
    for ( int i=0; i<counter; i++ ) {
      sample = data->volume * data->reverb->
tick( data->voicer->tick() );
      for ( unsigned int j=0; j<data->nWvOuts;
 j++ ) data->wvout[j]->tick(sample);
      if ( data->realtime )
        for ( int k=0; k<data->channels;
 k++ ) *samples++ = sample;
      nTicks--;
    }
    if ( nTicks == 0 ) break;

    // Process control messages.
    if ( data->haveMessage ) processMessage( data );
  }

  return 0;
}

int main( int argc, char *argv[] )
{
  TickData data;
  int i;
```

```
#if defined(__STK_REALTIME__)
  RtAudio *dac = 0;
#endif

  // If you want to change the
default sample rate (set in Stk.h), do
  // it before instantiating any
objects!  If the sample rate is
  // specified in the command line,
it will override this setting.
  Stk::setSampleRate( 44100.0 );

  // By default, warning messages
are not printed.  If we want to see
  // them, we need to specify that here.
  Stk::showWarnings( true );

  // Check the command-line
arguments for errors and to determine
  // the number of WvOut
objects to be instantiated (in utilities.cpp).
  data.nWvOuts = checkArgs(argc, argv);
  data.wvout = (WvOut **) calloc(data.nWvOuts,
 sizeof(WvOut *));

  // Instantiate the instrument(s) type
from the command-line argument
  // (in utilities.cpp).
  data.nVoices = countVoices(argc, argv);
  data.instrument =
(Instrmnt **) calloc(data.nVoices, sizeof(Instrmnt *));
  data.currentVoice =
 voiceByName(argv[1], &data.instrument[0]);
  if ( data.currentVoice < 0 ) {
    free( data.wvout );
    free( data.instrument );
    usage(argv[0]);
  }
  // If there was no
error allocating the first voice, we
should be fine for more.
  for ( i=1; i<data.nVoices; i++ )
    voiceByName(argv[1], &data.instrument[i]);
```

```
  data.voicer = (Voicer *) new Voicer( data.nVoices );
  for ( i=0; i<data.nVoices; i++ )
    data.voicer->addInstrument( data.instrument[i], i+1 );

  // Parse the command-line
flags, instantiate WvOut objects, and
  // instantiate the input
message controller (in utilities.cpp).
  try {
    data.realtime = parseArgs(argc,
argv, data.wvout, data.messager);
  }
  catch (StkError &) {
    goto cleanup;
  }

  // If realtime output, allocate the dac here.
#if defined(__STK_REALTIME__)
  if ( data.realtime ) {
    RtAudioFormat format = ( sizeof(StkFloat) ==
 8 ) ? RTAUDIO_FLOAT64 : RTAUDIO_FLOAT32;
    int bufferSize = RT_BUFFER_SIZE;
    try {
      dac = new RtAudio(0, data.channels,
 0, 0, format, (int)Stk::sampleRate(), &bufferSize, 4);
    }
    catch (RtError& error) {
      error.printMessage();
      goto cleanup;
    }
  }
#endif

  // Set the reverb parameters.
  data.reverb = new PRCRev( data.t60 );
  data.reverb->setEffectMix(0.2);

  // Install an interrupt handler function.
        (void) signal(SIGINT, finish);

  // If realtime output, set
our callback function and start the dac.
#if defined(__STK_REALTIME__)
  if ( data.realtime ) {
```

```
    try {
      dac->setStreamCallback(&tick, (void *)&data);
      dac->startStream();
    }
    catch (RtError &error) {
      error.printMessage();
      goto cleanup;
    }
  }
#endif

  // Setup finished.
  while ( !done ) {
#if defined(__STK_REALTIME__)
    if ( data.realtime )
      // Periodically check "done" status.
      Stk::sleep( 200 );
    else
#endif
      // Call the "tick" function to process data.
      tick( NULL, 256, (void *)&data );
  }

  // Shut down the callback and output stream.
#if defined(__STK_REALTIME__)
  if ( data.realtime ) {
    try {
      dac->cancelStreamCallback();
      dac->closeStream();
    }
    catch (RtError& error) {
      error.printMessage();
    }
  }
#endif

 cleanup:

  for ( i=0; i<(int)data.nWvOuts; i++ ) delete data.wvout[i];
  free( data.wvout );

#if defined(__STK_REALTIME__)
  delete dac;
#endif
```

```
  delete data.reverb;
  delete data.voicer;

  for ( i=0; i<data.nVoices; i++ ) delete data.instrument[i];
  free( data.instrument );

  return 0;
}
```

Bibliografía

[1] Vesa Välimäki y Tapio Takala. *Virtual musical instruments - natural sound using physical models.* Helsinki University of Technology, `www.imi.aau.dk/~sts/classes/ia06/valimakitakala.pdf`, 1996.

[2] Julius O. Smith III. *Physical Modeling using digital waveguides.* Stanford University, `http://www-ccrma.stanford.edu/~jos/pmudw/pmudw.pdf`, 1992.

[3] Synthesis Toolkit homepage: `http://ccrma.stanford.edu/software/stk/`.

[4] TAO project homepage: `http://web.ukonline.co.uk/taosynth/`.

[5] Puredata homepage: `http://puredata.info/`.

[6] GTK+, GIMP ToolKit homepage: `http://www.gtk.org`.

[7] Matti Vihola, *Sound Synthesis Methods.* Tampere University of Technology, `http://www.cs.tut.fi/sgn/arg/synteesi/vihola.pdf`, 2001.

[8] Dave Benson. *Music: A Mathematical offering.* Department of Mathematical Sciences, University of Aberdeen, Scotland, UK, `http://www.maths.abdn.ac.uk/~bensondj/html/music.pdf`, 2006.

[9] *Chebyshev Polinomials.* Report from the Department of Mathematics, California State University, Fullerton, `http://math.fullerton.edu/mathews/n2003/ChebyshevPolyMod.html`.

[10] *Low-pass filter.* Wikipedia, the free encyclopedia, `http://en.wikipedia.org/wiki/Low-pass_filter`.

[11] Antares home page: `http://www.antarestech.com`

[12] Chuck homepage: `http://chuck.cs.princeton.edu/`

[13] Soundlab princeton homepage: `http://soundlab.cs.princeton.edu/research`

Los abajo firmantes autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Fdo: Pedro Javier Calle Cantalapiedra

Fdo: Javier Chávarri Álvarez

Fdo: Fernando Iglesias Pulido